A Systematic Approach to Automatically Generate Multiple Semantically Equivalent Program Versions

Sri Hari Krishna Narayanan, Mahmut Kandemir
Department of Computer Science and Engineering
The Pennsylvania State University
{snarayan,kandemir}@cse.psu.edu

Abstract

Classic methods to overcome software faults include design diversity that involves creating multiple versions of an application. However, design diverse techniques typically require a staggering investment of time and manpower. There is also no guarantee that the multiple versions are correct or equivalent. This paper presents a novel approach that addresses the above problems, by automatically producing multiple, semantically equivalent copies for a given array/loop-based application. The copies, when used within the framework of common design diverse techniques, provide a high degree of software fault tolerance at practically no additional cost. In this paper, we also apply our automated version generation approach to detect the occurrence of soft errors during the execution of an application.

This work has been supported in part by NSF grants # 0720645, # 0702519 and support from the Gigascale Systems Research Focus Center, one of the five research centers funded under SRC's Focus Center Research Program.

A Systematic Approach to Automatically Generate Multiple Semantically Equivalent Program Versions

Sri Hari Krishna Narayanan and Mahmut Kandemir

Department of Computer Science and Engineering
The Pennsylvania State University, University Park, PA 16802, USA
{ snarayan, kandemir} @ cse.psu.edu

Abstract. Classic methods to overcome software faults include design diversity that involves creating multiple versions of an application. However, design diverse techniques typically require a staggering investment of time and manpower. There is also no guarantee that the multiple versions are correct or equivalent. This paper presents a novel approach that addresses the above problems, by automatically producing multiple, semantically equivalent copies for a given array/loop-based application. The copies, when used within the framework of common design diverse techniques, provide a high degree of software fault tolerance at practically no additional cost. In this paper, we also apply our automated version generation approach to detect the occurrence of soft errors during the execution of an application.

1 Introduction

Design diversity is a technique used for achieving a certain degree of fault tolerance in software engineering [1–5]. Since exact copies of a given program cannot always improve fault tolerance, creating multiple, different copies is essential [6]. However, this is not a trivial task as independently designing different versions of the same application software can take a lot of time and resources, most of which is spent verifying that these versions are indeed semantically equivalent and they exhibit certain diversity which helps us catch design errors as much as possible (e.g., by minimizing the causes for identical errors). The problem becomes more severe if a large number of versions are required.

Automatically generating different versions of a given program can be useful in two aspects, provided that the versions generated are sufficiently diverse for catching the types of errors targeted. First, design time and cost can be dramatically reduced as a result of automation. Second, since the versions are generated automatically, we can be sure that they are semantically equivalent save for the errors of interest. However,

This work is supported in part by NSF grants # 0720645 , # 0702519 and support from the Gigascale Systems Research Focus Center, one of the five research centers funded under SRCs Focus Center Research Program. The authors would like to thank the anonymous reviewers for their helpful remarks. The authors would like to thank Seung Woo Son and Shiva Prasad Kasiviswanathan for their suggestions. Finally, the authors would like to thank, our shepherd, Dr. Erhard Plödereder who helped finalize the paper.

as mentioned earlier, these versions should be sufficiently different from each other, depending on the types of errors targeted.

Numerical applications which make extensive use of arrays and nested loops are good candidates for automatic version generation as they are amenable to be analyzed and restructured by optimizing compilers. Current compilers restructure such applications to optimize data locality and improving loop-level parallelism as well as for other reasons [7–10]. The main stumbling block to full fledged re-ordering of computations are data dependences in the program code.

The main contribution of this paper is a tool that generates different versions of a numerical application automatically *a priori*. The tool generates these versions by restructuring the given application code in a systematic fashion using the concept of *data tiles*. A data tile is a portion of an array which may be manipulated by the application. Hence, an array can be thought of as a series of data tiles. Given such a series of data tiles, of a particular size and shape, we can generate a new version of the code by restructuring the code in such a fashion that the accesses to each tile are completed before moving to the next tile. As a result, computations are performed on a per tile basis. Therefore, a different tile shape or a different order of tiles (to the extent allowed by data) gives an entirely different version of the application, thereby contributing to diversity. In this paper, we also present a method for selecting the tile shapes as well as method to systematically reorder them based on the number of versions required.

We apply our tool to the emergent architectural challenge of soft errors. Soft errors are a form of transient errors that occur when charged neutrons strike logic devices which hold charges to indicate the bit that they represent [11–14]. A neutron strike can change the charge held on the device either by charging or discharging it. This change in charge can lead to a bit flip in memory or logic components of the system which can affect the end results generated by the application. We show how the tool can be used to detect errors that remain undetected by a state of the art architectural recovery approach.

The remainder of this paper is organized as follows. Section 2 presents the theory behind the proposed approach. Section 3 presents implementation details of our tool as well as results obtained using a scientific benchmark. Section 4 concludes the paper by summarizing our major contributions and giving a brief outline of the planned future work.

2 Detailed Analysis

This section explains the details of the approach proposed to automatically create the multiple versions of a given array/loop based application. Our goal is to obtain different (but semantically equivalent) versions of a given code fragment by restructuring the fragment based on a data tile shape. The input to our approach is a code fragment that consists of the series of loop nests and the data array(s) that is accessed in the fragment. The loop nests in the fragment contain expressions, called array references, that access locations within the array. Figure 1(a) shows an example code fragment and the array being accessed in the loop nests.

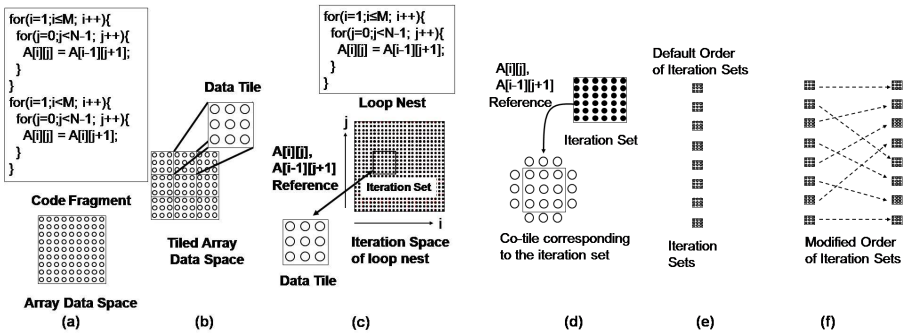

Fig. 1. (a) A code fragment. (b) Data tiles formed from a seed tile. (c) Iteration set that accesses the data in a data tile. (d) Co-tile identification. (e) Default order of iteration sets. (f) New order of iteration sets, as a result of restructuring.

Our approach first creates a *seed tile* which is a uniquely shaped subsection of the array (selection of a seed tile is detailed in Section 2.7). Using this seed tile as a template, we logically divide the array into multiple sections called *data tiles* as shown in Figure 1(b). In the following paragraphs we discuss what is performed on a particular data tile.

In the next stage shown in 1(c), we identify for each loop nest the array references that accesses locations within the data tile. Then, for each loop nest, we use these references to determine the set of iterations that access this particular data tile. The iterations from a loop nest that are associated with a particular data tile are called the *iteration set* of that data tile with respect to that loop nest.

Now, let us consider the case for a particular iteration set associated with a data tile. It is possible that these iterations access array locations outside the data tile as well. These external locations are called the *extra tile*, and the original data tile and the extra tile are collectively referred to as the *co-tile*. Figure 1(d) shows the co-tile corresponding to an iteration set.

Our idea is to first identify, for each combination of data tile and loop nest, the associated iteration set. Once we have the iteration set corresponding to a data tile and loop nest, we can execute all the computations that should take place on that pair. The original code can therefore be thought of as the default order of iteration sets shown in Figure 1(e). Next, in order to create new codes, we systematically re-order the iteration sets to create multiple different sequences as shown in Figure 1(f). Each unique order of iteration sets leads to a unique version of the code. Such a re-ordering is legal provided that data dependences do not exist between iteration sets. Data dependences, impose an ordering constraint on the iteration sets and prevent full fledged re-ordering. If dependences do exist between the iteration sets, we explore other data tile shapes to arrive at a dependence free group of iteration sets.

The rest of this section details our approach. After presenting basic definitions in Section 2.1, Section 2.2 presents our method of forming data tiles. Section 2.3 shows how iteration sets and co-tiles are calculated. Our algorithm to detect dependences (legality requirements) are presented in Section 2.4. Section 2.5 shows how the iteration sets are systematically re-ordered. Section 2.6 presents the overall algorithm used to cre-

ate multiple versions of code. Section 2.7 discusses how data tiles of different shapes and sizes are created, and Section 2.8 explains how we deal with code that accesses multiple arrays.

2.1 Basic Definitions

This subsection presents important definitions that we use to formalize our approach.

- Program : A program source code fragment is represented as $\mathcal{P} = \{\mathcal{N}, \mathcal{A}\}$, where $\mathcal{N}$ is a list of loop nests and $\mathcal{A}$ is the set of arrays declared in $\mathcal{P}$ that are accessed in $\mathcal{N}$. Figure 2 shows the benchmark source code fragment employed.
- Array : An array $\mathcal{A}_a$ is described by its dimensions, δ, and the extent (size) in each dimension, γ, $\mathcal{A}_a = \{\delta, \gamma\}$. For example, the array DW defined in the code fragment in Figure 2 can be expressed as $DW = \{3, \{10, 10, 4\}\}$ in our framework.
- Loop Nest : A loop nest $\mathcal{N}_i$, is represented as $\{\alpha, \mathcal{A}_\mathcal{N}, \boldsymbol{\mathcal{I}}, \boldsymbol{\mathcal{L}}, \boldsymbol{\mathcal{U}}, \boldsymbol{\mathcal{S}}, \psi\}$, where α is the number of loops in the nest and $\boldsymbol{\mathcal{L}}, \boldsymbol{\mathcal{U}}$, and $\boldsymbol{\mathcal{S}}$ are vectors that give, respectively, the values of the lower limit, upper limit, and the step of the loop index variables which are given in $\boldsymbol{\mathcal{I}}$. It is assumed that at compile time all the values of these vectors are known. The body of the loop nest is represented by ψ. The arrays accessed within $\mathcal{N}_i$ are represented as $\mathcal{A}_{\mathcal{N}_i}$ where $\mathcal{A}_{\mathcal{N}_i} \subseteq \mathcal{A}$, i.e., each loop nest typically accesses a subset of the arrays declared in the program code. For example, the second loop nest in Figure 2 can be represented as

$$N_1 = \left\{ 3, DW, \begin{bmatrix} N \\ J \\ I \end{bmatrix}, \begin{bmatrix} 1 \\ 2 \\ 2 \end{bmatrix}, \begin{bmatrix} 4 \\ 10 \\ 10 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \psi \right\}.$$

- Loop Body : A loop body is made of a series of statements which use the references to the arrays $\mathcal{A}$ declared in $\mathcal{P}$. Consequently, loop body ψ can be expressed as a set of references.
- Iteration : For a loop nest, $\mathcal{N}_n$, an iteration is a particular combination of legal values that its index variables in $\boldsymbol{\mathcal{I}}$ can assume. It is expressed as $\boldsymbol{\mathcal{I}}_\sigma$, and it represents an execution of the loop body.
- Iteration Space : The iteration space of a loop nest $\mathcal{N}_i$ is the set of all iterations in the loop nest.
- Data Space : The data space of a data structure (e.g., an array) are all the individual memory locations that form the data structure in question.
- Reference : It is an element of ψ expressed as ($\psi_p^{r/w} = \{\mathcal{N}_n, \mathcal{A}_\mathcal{A}, L, \boldsymbol{o}\}$). It is an affine relation from the iteration space of a loop nest $\mathcal{N}_n = \{\alpha, \mathcal{A}_n, \boldsymbol{\mathcal{I}}, \boldsymbol{\mathcal{L}}, \boldsymbol{\mathcal{U}}, \boldsymbol{\mathcal{S}}, \psi\}$ to the data space of an array ($\mathcal{A}_a = \{\delta, \gamma\}$). From compiler theory [7], it is known that this relation can be described by $L\boldsymbol{i} + \boldsymbol{o}$ where $\boldsymbol{i}$ is a vector that captures the loop indices of $\mathcal{N}$, L is a matrix of size $\delta * \alpha$, and $\boldsymbol{o}$ is an offset displacement vector. As an example, the reference $A[i + j - 1][j + 2]$ is represented by

$$\psi_p^{r/w} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} -1 \\ 2 \end{bmatrix}.$$

  A reference within the body of a loop nest helps us calculate the locations of an array that the loop nest accesses. Further, a reference can be a read reference, which

```
int DW[10][10][4];

for (N=1;N<=4;N++) {
 for (J=2;J<=10;J++)
  DW[1][J][N] = 0;
}

for (N=1;N<=4;N++) {
 for (J=2;J<=10;J++)
  for (I=2;I<=10;I++)
   DW[I][J][N] = DW[I][J][N]
                 -R*(DW[I][J][N]
                 -DW[I-1][J][N]);
}

for (N=1;N<=4;N++) {
 for (J=2;J<=10;J++)
  DW[10][J][N] = T1*DW[10][J][N];
}

for (N=1;N<=4;N++) {
 for(II=3; II<= 9; II++)
  for (J=2;J<=10;J++)
   DW[II][J][N] = DW[II][J][N]
                  -R*(DW[II][J][N]
                  -DW[II+1][J][N]);
}
```

Fig. 2. A code fragment with four loop nests and an array.

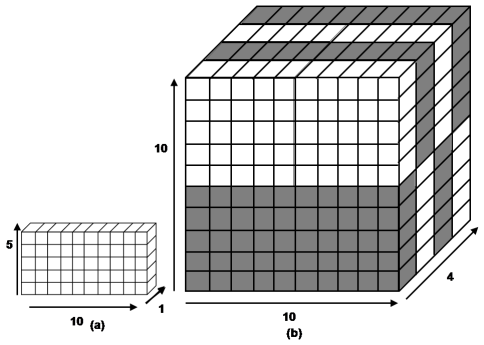

Fig. 3. (a) Seed tile for the array DW in the code fragment of Figure 2. (b) The array DW divided into multiple tiles using the seed tile.

.

means that an array location is read from, or a write reference, which means that an array location is written to. This is identified by attaching a r/w superscript to the reference. Hence, $\psi_p^r(\mathcal{N}_n)$ represents the set of all array locations read by the reference in loop nest $\mathcal{N}_n$.

2.2 Data Tile Formation

In this paper, we use the concept of data space tiling to logically divide the data space of an array into multiple sections. This subsection provides the theoretical basis and the algorithm used to perform tiling.

- Data Tile : A data tile $D_{\mathcal{A}_a,\boldsymbol{\mathcal{L}},\boldsymbol{\mathcal{U}}}$ is a regular subpart (region) of the array $\mathcal{A}_a$. The size of the data tile in each dimension is given by the difference between $\boldsymbol{\mathcal{L}}$ and $\boldsymbol{\mathcal{U}}$ plus 1. It is assumed that the size of a data tile is not zero in any dimension. Based on the definition of a data tile, data space of $D_{\mathcal{A}_a,\boldsymbol{\mathcal{L}},\boldsymbol{\mathcal{U}}}$ can be formally expressed as follows:

$$D_{\mathcal{A}_a,\boldsymbol{\mathcal{L}},\boldsymbol{\mathcal{U}}} = \{\{d_1, d_2..d_\delta\} \mid \boldsymbol{\mathcal{L}}_1 \leq d_1 \leq \boldsymbol{\mathcal{U}}_1 \\ \&\& \quad \boldsymbol{\mathcal{L}}_2 \leq d_2 \leq \boldsymbol{\mathcal{U}}_2 \ \ldots \ \&\& \quad \boldsymbol{\mathcal{L}}_\delta \leq d_\delta \leq \boldsymbol{\mathcal{U}}_\delta\}$$

- Seed Data Tile : A data tile, $D_{\mathcal{A}_a,\boldsymbol{\mathcal{L}},\boldsymbol{\mathcal{U}}}$, is described as a seed data tile if $\boldsymbol{\mathcal{L}} = \mathbf{0}$. This tile is used (as a template) to partition the array $\mathcal{A}_a$ into further tiles. As an example, Figure 3(a) shows a seed tile for the array DW that is defined in Figure 2, and Figure 3(b) illustrates how DW is partitioned into multiple tiles using this seed tile. This partitioning is outlined in Algorithm 1. Multiple seed tiles can simply formed by changing the values of the entries of $\boldsymbol{\mathcal{U}}$. By supplying different seed tiles as input to Algorithm 1, we are able to split an array into differently shaped tiles.

Algorithm 1 $DataTile(D_{\mathcal{A}_a,\boldsymbol{\mathcal{L}},\boldsymbol{\mathcal{U}}})$

```
1:  Tile_list := ∅
2:  for i_δ = 1 to γ_δ by U[δ] do
3:    L'[δ] := i_δ
4:    U'[δ] := min(i_δ + U[δ] − 1, γ_δ)
5:    for i_{δ−1} = 1 to γ_{δ−1} by U[δ − 1] do
6:      L'[δ − 1] := i_{δ−1}
7:      U'[δ − 1] := min(i_{δ−1} + U[δ − 1] − 1,
8:                                        γ_{δ−1})
9:      .
10:     .
11:     for i_1 = 1 to γ_1 by U[1] do
12:       L'[1] := i_1
13:       U'[1] := min(i_1 + U[1] − 1, γ_1)
14:       Tile := D_{A_a,L',U'}
15:       Tile_list := Tile_list ⋃ Tile
16:     end for
17:   end for
18: end for
19: Return Tile_list
```

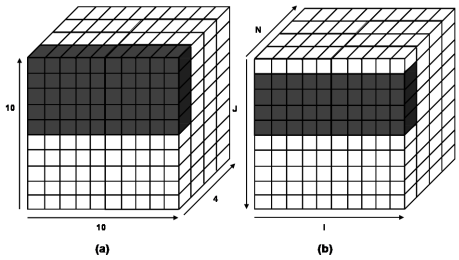

Fig. 4. The iteration set corresponding a data tile in the array DW (accessed by the code fragment in Figure 2) and the second loop nest in the code fragment.

2.3 Iteration Set and Co-tile Formation

An iteration set is associated with a loop nest $\mathcal{N}_n$, and a data tile $D_{\mathcal{A}_a,\boldsymbol{\mathcal{L}},\boldsymbol{\mathcal{U}}}$. It is the subset of the iteration space of $\mathcal{N}_n$, in which the elements (iterations) have the property that $\psi_p^{r/w}(\boldsymbol{\mathcal{I}}_\sigma) \in D_{\mathcal{A}_a,\boldsymbol{\mathcal{L}},\boldsymbol{\mathcal{U}}}$. That is, it is the set of all iterations in a particular loop nest that accesses the locations in a given data tile. We can calculate the iteration set $I(D_{\mathcal{A}_a,\boldsymbol{\mathcal{L}},\boldsymbol{\mathcal{U}}},\mathcal{N}_n)$ of data tile $D_{\mathcal{A}_a,\boldsymbol{\mathcal{L}},\boldsymbol{\mathcal{U}}}$ and loop nest $\mathcal{N}_n$ as

$$I(D_{\mathcal{A}_a,\boldsymbol{\mathcal{L}},\boldsymbol{\mathcal{U}}},\mathcal{N}_n) = \bigcup_{\psi_p^{r/w}\in\psi} \bigcup_{\mathcal{I}_\sigma\in\mathcal{N}_n} \{ \ \mathcal{I}_\sigma \ | \ \{\psi_p^{r/w}(\mathcal{I}_\sigma) \cap D_{\mathcal{A}_a,\boldsymbol{\mathcal{L}},\boldsymbol{\mathcal{U}}}\} \neq \emptyset \ \}. \quad (1)$$

Figure 4 shows the iteration set corresponding to the data tile of the array DW and the second loop nest in the code given in Figure 2. It is possible that the iteration set $I(D_{\mathcal{A}_a,\boldsymbol{\mathcal{L}},\boldsymbol{\mathcal{U}}},\mathcal{N}_n)$ accesses locations in the array $\mathcal{A}_a$ that lie outside the data tile, $D_{\mathcal{A}_a,\boldsymbol{\mathcal{L}},\boldsymbol{\mathcal{U}}}$. In other words, $(\bigcup_{\psi_p} \psi_p(I(D_{\mathcal{A}_a,\boldsymbol{\mathcal{L}},\boldsymbol{\mathcal{U}}},\mathcal{N}_n))) - D_{\mathcal{A}_a,\boldsymbol{\mathcal{L}},\boldsymbol{\mathcal{U}}} \neq \emptyset$ may be true.

Recall that our overall goal is to capture all the computations that need to be performed by a loop nest on a data tile. As a consequence, we need to express the extra locations that are accessed by the iteration set. As mentioned earlier, the extra locations and the original data tile together are called the co-tile of the iteration set and is given by:

$$C_{D_{\mathcal{A}_a,\boldsymbol{\mathcal{L}},\boldsymbol{\mathcal{U}},\mathcal{N}_i}} = \bigcup_{\forall\psi_p\in\mathcal{N}_n} \psi_p(I(D_{\mathcal{A}_a,\boldsymbol{\mathcal{L}},\boldsymbol{\mathcal{U}}},\mathcal{N}_n)) \quad (2)$$

Using the formulation for iteration set in Equation (1), the formulation for a co-tile given in Equation (2) and the list of all data tiles generated by Algorithm 1, we can now generate a list of all iteration set/co-tile pairs. The default list of pairs describes the default program behavior (i.e., without any restructuring). It is this behavior that we want to change while maintaining the same semantics as the original code.

Algorithm 2 $DependenceDetector(Tile\_list)$

```
1:  Dep_Array := 0
2:  for all D_m ∈ Tile_list do
3:    for all N_i ∈ N do
4:      calculate I_{D_m,N_i}
5:    end for
6:  end for
7:  for all D_m ∈ Tile_list do
8:    for all N_i ∈ N do
9:      for all D_n ∈ Tile_list do
10:       for all N_j ∈ N do
11:         if {(⋃_{ψ^w_p} ψ^w_p(I_{D_m,N_i})) ⋂ (⋃_{ψ^r_{p'}} ψ^r_{p'}(I_{D_n,N_j}))} ≠ ∅||
            {(⋃_{ψ^r_p} ψ^r_p(I_{D_m,N_i})) ⋂ (⋃_{ψ^w_{p'}} ψ^w_{p'}(I_{D_n,N_j}))} ≠ ∅||
            {(⋃_{ψ^w_p} ψ^w_p(I_{D_m,N_i})) ⋂ (⋃_{ψ^w_{p'}} ψ^w_{p'}(I_{D_n,N_j}))} ≠ ∅ then
12:           Dep_Array_{m,i,n,j} := 1
13:         end if
14:       end for
15:     end for
16:   end for
17: end for
18: Return Dep_Array
```

2.4 Data Dependences Across Iteration Sets

All iterations in the given program fragment are executed in a default order called the program order. This program order can be extended to the pairs of iteration sets and co-tiles. In order to change the code, the execution of iteration sets must be re-ordered. A fundamental restriction on whether we can re-order the iteration sets are ordering relations among them, which are also known as *data dependences*.

The execution order of any two iterations can be arbitrary with respect to each other as long as these two iterations do not have any data dependence between them. A data dependence exists between two iterations within a loop nest if one iteration reads a value of a variable computed by another iteration or if both iterations compute the value of the same variable [7].

Consequently, in order to re-order any two iteration sets, there should not be any data dependence there between them. Furthermore, if we want to arbitrarily re-order all the iteration sets, there should not be any data dependence between any two iteration sets. Otherwise, it is possible that the wrong data is read by one iteration set or written by another iteration set. The rest of this sub-section presents our algorithm to detect data dependences between iteration set and co-tile pairs. This analysis is different from conventional data dependence analysis as we perform it at an iteration set and co-tile granularity.

Formally, two iterations $\mathcal{I}_\sigma$ and $\mathcal{I}'_\sigma$ within a nest $\mathcal{N}_n$ have a data dependence between them if and only if

$$\psi^r_p(\mathcal{I}_\sigma) = \psi^w_{p'}(\mathcal{I}'_\sigma) || \psi^w_p(\mathcal{I}_\sigma) = \psi^r_{p'}(\mathcal{I}'_\sigma) || \psi^w_p(\mathcal{I}_\sigma) = \psi^w_{p'}(\mathcal{I}'_\sigma) \quad (3)$$

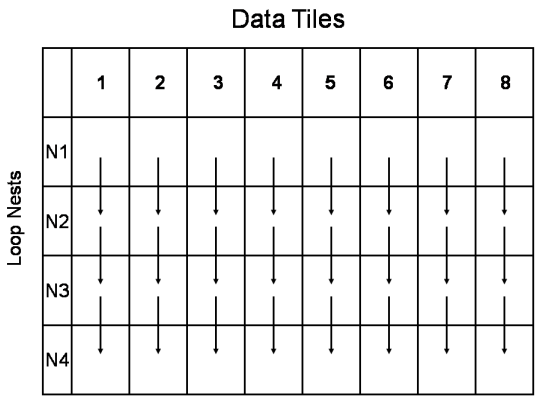

Fig. 5. Arrows indicate the data dependence between iteration sets formed by loop nests in Figure 2 and data tiles formed using the seed tile in Figure 3(a).

is true, where $\psi_p^{r/w}$ and $\psi_{p'}^{r/w}$ are two references that appear in $\mathcal{N}_n$.

This formulation can be extended to iteration sets and the co-tiles that are accessed in them. In the context of our paper, a dependence is said to exist between two iteration sets if and only if,

$$\begin{aligned}
&\{(\bigcup_{\psi_p^w} \psi_p^w(I(D_{\mathcal{A}_a,\boldsymbol{\mathcal{L}},\boldsymbol{\mathcal{U}}},\mathcal{N}_n))) \bigcap (\bigcup_{\psi_{p'}^r} \psi_{p'}^r(I(D_{\mathcal{A}_a,\boldsymbol{\mathcal{L}'},\boldsymbol{\mathcal{U}'}},\mathcal{N}_{n'})))\} \neq \emptyset || \\
&\{(\bigcup_{\psi_p^r} \psi_p^r(I(D_{\mathcal{A}_a,\boldsymbol{\mathcal{L}},\boldsymbol{\mathcal{U}}},\mathcal{N}_n))) \bigcap (\bigcup_{\psi_{p'}^w} \psi_{p'}^w(I(D_{\mathcal{A}_a,\boldsymbol{\mathcal{L}'},\boldsymbol{\mathcal{U}'}},\mathcal{N}_{n'})))\} \neq \emptyset || \\
&\{(\bigcup_{\psi_p^w} \psi_p^w(I(D_{\mathcal{A}_a,\boldsymbol{\mathcal{L}},\boldsymbol{\mathcal{U}}},\mathcal{N}_n))) \bigcap (\bigcup_{\psi_{p'}^w} \psi_{p'}^w(I(D_{\mathcal{A}_a,\boldsymbol{\mathcal{L}'},\boldsymbol{\mathcal{U}'}},\mathcal{N}_{n'})))\} \neq \emptyset
\end{aligned} \tag{4}$$

is true.

Based on Equation (4), Algorithm 2 detects the data dependences between the iteration sets formed from a list of data tiles. As we are not interested in re-ordering the iterations within an iteration set, dependence detection is performed at the level of loop nest granularity. The algorithm sets $Dep\_Array[m,i,n,j]$ to 1 if a dependence exists between the iteration set $I_{D_m,\mathcal{N}_i}$ and the iteration set $I_{D_n,\mathcal{N}_j}$, where D_m and D_n are data tiles created by Algorithm 1. For two iteration sets associated with the same loop nest, the dependence flows from the iteration set that contains the earlier iterations to the other iteration set. Let us now discuss what the matrix Dep_Array represents. The dependence relations between iteration sets can be described by a graph in which the nodes are the individual iteration sets. A directed edge from the node that represents iteration set $I_{D_m,\mathcal{N}_i}$ to the node that represents $I_{D_n,\mathcal{N}_j}$ means that $I_{D_n,\mathcal{N}_j}$ is dependent on $I_{D_m,\mathcal{N}_i}$. Consequently a node that represents an iteration set that is independent of all other iteration sets has a fan-in value of zero in this graph. Given these observations, we can conclude that the matrix Dep_Array is simply the representation of this

Position	Code Version							
	1	2	3	4	5	6	7	8
1	1	2	3	4	5	6	7	8
2	2	3	4	5	6	7	8	1
3	3	4	5	6	7	8	1	2
4	4	5	6	7	8	1	2	3
5	5	6	7	8	1	2	3	4
6	6	7	8	1	2	3	4	5
7	7	8	1	2	3	4	5	6
8	8	1	2	3	4	5	6	7

Fig. 6. The different orders of iteration sets in the different versions of the code

Algorithm 3 $VersionGenerator(P_o, V)$

```
1: Generate Seed Tile
2: Create Iteration Sets and Partitions
3: Verify dependences
4: while Dependences exist do
5:    if Generate New Seed Tile() == failure then
6:       Return Error
7:    end if
8:    Create Iteration Sets and Partitions
9:    Verify dependences
10: end while
11: Create V Versions
```

graph in an adjacency matrix form. The dependence relations between iteration sets is represented pictorially in Figure 5.

At this point, we have generated a list of iteration sets which when executed individually perform all the computations that should be performed on a particular data tile by the associated loop nest. However, it is possible that two iteration sets, $I_{D_n,\mathcal{N}_j}$ and $I_{D_{n'},\mathcal{N}_j}$, which are associated with the same nest and have a dependence between them, might intersect. That is, some iterations may belong to both $I_{D_n,\mathcal{N}_j}$ and $I_{D_{n'},\mathcal{N}_j}$. In order to produce code that is semantically identical to the original code, the intersecting iterations need to be associated with only one of the iteration sets. Assuming that the iteration set $I_{D_{n'},\mathcal{N}_j}$ is dependent on $I_{D_n,\mathcal{N}_j}$, the intersecting iterations are executed by $I_{D_{n'},\mathcal{N}_j}$. That is, $I_{D_n,\mathcal{N}_j}$ is set to $I_{D_n,\mathcal{N}_j} - (I_{D_n,\mathcal{N}_j} \cap I_{D_{n'},\mathcal{N}_j})$.

2.5 Re-ordering Iteration Sets

The key requirement for full re-ordering of iteration sets is that there should be no data dependence at all between iteration sets. However, this behavior is not exhibited by most real applications. Therefore, we relax this requirement and allow reordering when the only dependences are between iteration sets corresponding to the same data tile. That is, directed edges of the form, $I_{D_n,\mathcal{N}_i}$ to $I_{D_n,\mathcal{N}_j}$ which represents data dependences between iteration sets associated with the same data tile are allowed. Once this condition has been satisfied, we first group all the iteration sets associated with each tile. Then, we partition the groups of iteration sets into V groups, where V is the number of versions of code that are required and number each partition from 1 to V. We use this numbering to create a circular sequence over all the iteration set partitions. That is, to create the i^{th} version of the code the order of iteration set partitions is : i, $i+1$ ... $V-1$, V, 1, 2....$i-2$, $i-1$. Figure 6 presents the orders of partitions when V is 8.

2.6 Generating Multiple Versions

This section describes Algorithm 3 to create the multiple versions of an input program. The input to the algorithm is the original program P_o and number of versions, V, of

the code that are desired. In order to create a semantically equivalent version of P_o, a new seed element (that has not been used previously) is formed. Then, using this seed element, the data space of P_o is broken up into further data tiles.

Using these data tiles and the loop nests in P_o, the dependence graph between the iteration sets that correspond to these data tiles is created. If there are no dependences between iteration sets corresponding to different data tiles, then the different versions of the code are created using orders as explained in Section 2.5. If however, dependences do exist, a new seed tile is used. If no satisfactory seed tile can be found, an error is reported. In order to generate the actual code, we rely on the Omega Library [15] which is a polyhedral tool in which iteration spaces can be described using Presburger arithmetic [16]. Given the description and order of the iteration tiles, the *codegen* utility of the Omega Library is used to generate the actual loop nests. Once the loops have been generated, they are combined so that the generated code is as compact as possible. However, the combining is done such that the order between the iteration sets remains the same. In fact, the combining method simply generates loops that iterate over the partitions of iteration sets. A portion of the semantically equivalent version of the code corresponding to one data tile is shown in Figure **??**.

2.7 Data Tile Selection

So far we have ignored the problem of generating the actual seed tiles which divide the array data space into its component tiles.

The potential space to explore in order to select appropriate seed tiles is vast. We first trim this space by considering only those tiles whose boundaries are parallel to the axes of the array that is being tiled. The rationale behind this is that the output codes generated using such tiles tend to be simpler that those generated using arbitrary tiles. That is, if the array is δ-dimensional, the seed is shaped regularly, and the references from the loop nest to the array are through *affine* expressions; then iteration sets that access the data tiles are regular in shape.

```
int DW[10][10][4];

for (J=2;J<=5;J++)
  DW[1][J][1] = 0;

for (J=2;J<=5;J++)
  for (I=2;I<=10;I++)
    DW[I][J][1] = DW[I][J][1]  -R*(DW[I][J][1]
                               -DW[I-1][J][1]);

for (J=2;J<=5;J++)
  DW[10][J][1] = T1*DW[10][J][1];

for(II=3; II<= 9; II++)
  for (J=2;J<=5;J++)
    DW[II][J][1] = DW[II][J][1]  -R*(DW[II][J][1]
                                 -DW[II+1][J][1]);
```

Fig. 7. The code generated for one data tile of the code given in Figure 2.

Further, as we require V different versions, we assume that the size of the seed tile should imply that there are V data tiles. This also implies that the iteration sets in an iteration set partition are all associated with the same data tile.

Let us consider a δ-dimensional array, $A[n_1, n_2, ..n_\delta]$ for which V unique seed tiles are required. As A is δ-dimensional, any seed tile of A, $S[s_1, s_2, ..s_\delta]$, is also δ-dimensional. Therefore, the problem of finding the values of $s_1, s_2, ..s_\delta$ which defines the shape of the seed tile translates into the problem of selecting an appropriate value of

s_i from the factors of n_i such that $\sum_i s_i = V$. As n_i is bounded by the array size large, the number of combinations from which $S[s_1, s_2, ..s_\delta]$ is selected is not very large.

2.8 Handling Multiple Arrays

Our formulation so far has assumed that the references in the loop nests (of the code for which we meant to generate multiple versions) access a single array. In order to extend our approach to multiple arrays, we first need to extend the concept of an iteration set. An iteration set is now associated with a loop nest as well as data tiles belonging to different arrays. As a result, the iteration set is expressed as $I_{\{D\},\mathcal{N}_j}$, where $\{D\}$ is the set of data tiles (from different arrays) which are accessed in that iteration set. If the loop nest associated with the iteration set does not contain references that access an array $\{D\}$ will not contain a data tile from that array. Consequently, dependences between two iteration sets can potentially occur if they both access a common location in any array used by the program.

Another consideration with multiple arrays is how the seed tile for each array is created. One approach is to simply have the same seed tile for each array. Another approach is to create different seed tiles for different arrays, where the shape of a seed tile associated with one array is independent of the seed tile chosen for another array. In yet another approach, a seed tile is created for a chosen array $\mathcal{A}_s$ with a fixed number of elements. The ratio of the elements in a seed tile for an array $\mathcal{A}_{s'}$ is fixed relative to the number of elements in a seed tile used for $\mathcal{A}_s$, and based on the number of elements in this seed tile, the shape of the tiles is determined. Consequently, by changing the number of elements in the seed tile used for $\mathcal{A}_s$ the seed tile used for $\mathcal{A}_{s'}$ is changed. As each approach potentially gives us different versions of code, the approach we choose depends on the number of versions that need to be created. The default approach used is the one in which each tile in each array is of the same shape.

3 Implementation and Experiments

While our automated approach can be useful in any scenario where multiple versions of the same code are needed, we focus on one particular scenario in this work. This section first describes the targeted scenario where our proposed approach is applied. It then illustrates the architecture of the tool that is created based on the approach. Finally, it describes the experiments conducted using the tool in the targeted scenario. As mentioned earlier, soft errors are a growing threat to the correct execution of an application [11–13]. A soft error is defined as an unwanted change in the state of a bit in a computer's component such as the memory system. It can result from particle strikes on logic devices which cause the bit represented by the device to flip. Increased scaling of technology has exacerbated this problem [14]. As result, the problem of soft errors has received considerable attention with many proposed hardware as well as software solutions. In chip multiprocessor (CMP) architectures, redundant-threading (RT) is one of the ways to overcome soft errors [17]. In an RT framework the same code is simultaneously executed across all the processors and periodically the results are compared to check if the computed results across the different threads agree. If they agree, it is

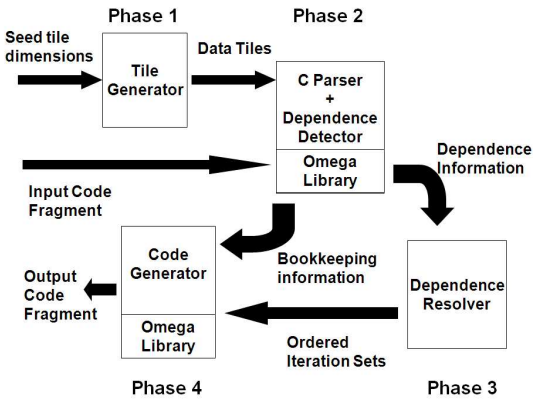

Fig. 8. Details of the flow within the tool. Phase 1 involves the creation of data tiles (Section 2.2) using a unique seed tile (Section 2.7). Phase 2 involves the parsing of the input code fragment, formation of iteration sets (Section 2.3), and detection of data dependences between them (Section 2.4). Phase 3 re-orders iteration sets (Section 2.5). Finally, phase 4 generates the output code fragment using the Omega Library (Section 2.6).

assumed that no error has occurred as only a single soft error is expected in any single thread and in any time frame. Another way is to run the code multiple times one after another and to check whether the results from each run agree with each other. Obviously, running each version simultaneously, if the resources are available, is the preferred option as it results in a faster finish time for the thread. The disadvantage is that in a CMP that is based on the shared memory concept, threads that operate simultaneously in the RT framework would read the same data from memory in close temporal proximity. Therefore, if a datum in memory is corrupted by a soft error, running the same code multiple times in parallel could result in the corrupted value being read by all threads. Such a read could result in the wrong result being computed and this error would remain undetected in current techniques. Although error correcting codes (ECC) have been proposed to overcome errors in the caches, ECC is not a viable solution in all computing systems due to the high costs it involves, especially form the power consumption angle [18, 19]. Furthermore, ECC would not catch multiple errors, which would be detected by our method.

We propose to use our automatic versioning algorithm to create multiple versions of the thread. These versions, when run in parallel, will access data in different temporal orders. Thus, the proposed approach will achieve temporal diversity without increasing the overall execution time. As a result, a particular datum which is corrupted at some time during the execution of the threads, could be accessed before corruption by one thread and after corruption by another. Therefore, it is possible that the changed value of the datum will be observable in the results of the different threads. Obviously, if the datum does not affect the end result, the proposed approach would perform exactly like the RT case and declare that no soft error has occurred. However, if that datum affects the end results, our approach is more likely to detect it. A tool was implemented based on the data tile based code restructuring approach (see Figure 8). This tools uses

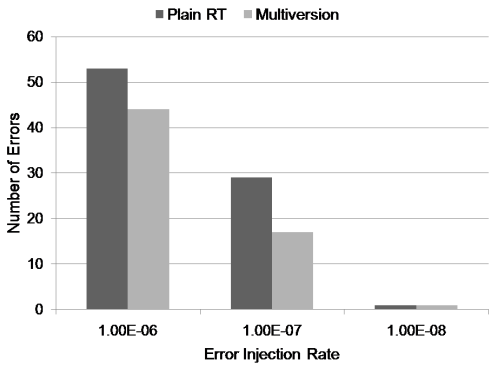

Fig. 9. The graph shows the number of errors in the array DW for different error injection rates using the default RT scheme and the proposed approach.

the Omega Library to evaluate the relations described in Section 2 and to generate the loops corresponding to the final relations using the Library's *codegen* utility on each relation one by one [20]. The tool was used to automatically create eight versions of the *tsf* benchmark shown in Figure 2 using the seed tile shown in Figure 3(a). Each version used a different order of iteration tiles shown in Figure 6. Therefore, each iteration set will execute at a particular time slot in at least one version. In an error free scenario, the different versions should generate the *same* results. However, in case of a soft error, two versions may differ in the results generated for a particular iteration set. In that case, the version that scheduled the iteration set earlier than the other is assumed to be the correct one. That is, the error is assumed to have occurred between the executions of the earlier set and set executed later.

We ran the original benchmark in conjunction with a fault injection module [21] to simulate execution under the soft error scenario. This setup was used to record the *stage* at which each error was injected and where in the memory space it occurred. Then, each automatically generated version of the code was run under the error injection mode using the previously recorded error occurrence and the results were compared with each other. A simple arbiter is used to reason about the results that are generated. If the results of any data tile in the automatically generated versions were different, the arbiter chose the results of the version in which the iteration tile corresponding to the data tile is executed earlier. In order to simulate RT, the errors recorded earlier are injected for each version, one at a time. At each stage, any error that is not injected into the memory is assumed to be caught, but any changes to the memory itself are allowed to propagate. Figure 9 shows the number of remaining errors in the proposed approach as compared to the standard RT approach (which uses the same version in each processor) for different injection rates. It can be seen that the proposed approach reduces the number of errors that affect the end result.

4 Concluding Remarks

This paper presents a tool that uses code restructuring techniques to automatically generate multiple semantically equivalent versions of a given numerical application that is

organized as a series of loops that access data in arrays. We created different versions of the code that differ in the order in which they access the data and used these different versions of the code to detect the occurrence of soft errors during the execution of the code. We believe that, this tool provides an inexpensive and automated method to enable fault tolerance to critical applications. Our planned future work includes developing more techniques to generate seed tiles easily and developing techniques to generate more compact code. We also plan to use our tool in other scenarios that benefit from multiple versions.

References

1. Avizienis, A.: On the implementation of nversion programming for software fault tolerance during execution. Proceedings of the IEEE **66**(10) (1978) 1109–1125
2. Elmendorf, W.: Fault-tolerant programming. In: FTCS-2. (1972) 79–83
3. Randell, B.: System structure for software fault tolerance. IEEE Trans. on Software Engineering **SE-1**(2) (1975) 220–232
4. Horning, J.J., al.: A program structure for error detection and recovery. In: Operating Systems, Proceedings of an Int. Symposium, Springer-Verlag (1974) 171–187
5. Pullum, L.: A new adjudicator for fault tolerant software applications correctly resulting in multiple solutions. In: Digital Avionics Systems Conference. (1993) 147–152
6. Pullum, L.L.: Software Fault Tolerance Techniques and Implementation. Artech House (2001)
7. Wolfe, M.: High Performance Compilers for Parallel Computing. Addison-Wesley (1996)
8. Wolfe, M.J.: Optimizing Supercompilers for Supercomputers. MIT Press (1990)
9. Kodukula, I., al.: Data-centric multi-level blocking. In: PLDI. (1997) 346–357
10. Kadayif, I., Kandemir, M.: Data space-oriented tiling for enhancing locality. Trans. on Embedded Computing Sys. **4**(2) (2005) 388–414
11. Michalak, S., Harris, K., Hengartner, N., Takala, B., Wender, S.: Predicting the number of fatal soft errors in los alamos national laboratory's asc q supercomputer. Device and Materials Reliability, IEEE Transactions on **5**(3) (Sept. 2005) 329–335
12. Wang, N., Quek, J., Rafacz, T., patel, S.: Characterizing the effects of transient faults on a high-performance processor pipeline. In: DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks. (2004) 61
13. Patel, J.: Characterization of soft errors caused by single event upsets in cmos processes. IEEE Trans. Dependable Secur. Comput. **1**(2) (2004) 128–143
14. Degalahal, V., Ramanarayanan, R., Vijaykrishnan, N., Xie, Y., Irwin, M.J.: The effect of threshold voltages on the soft error rate. In: International Symposium on Quality Electronic Design. (2004) 503–508
15. Kelly, W., al.: The omega calculator and library v1.1.0. Technical report, Dept. of CS, Univ. of Maryland (1996)
16. Kreisel, G., Krivine, J.L.: Elements of mathematical logic. North-Holland Pub. Co. (1967)
17. Reinhardt, S., Mukherjee, S.: Transient fault detection via simultaneous multithreading. SIGARCH Comput. Archit. News **28**(2) (2000) 25–36
18. Chen, C., Hsiao, M.: Error-correcting codes for semiconductor memory applications: a state of the art review. Reliable Computer Systems - Design and Evaluation (1992) 771–786
19. Pradhan, D.K., ed.: Fault-tolerant computer system design. (1996)
20. Kelly, W., al.: Code generation for multiple mappings. Technical report, Dept. of CS, Univ. of Maryland (1994)
21. S. Gurumurthi, A.P., Sivasubramaniam, A.: Sos: Using speculation for memory error detection. In: Workshop on High Performance Computing Reliability Issues. (2005)